Write a Compiler

David Beazley
http://www.dabeaz.com

May 2020

Deep Thought

Programming

```
int fact(int n) {
    int r = 1;
    while (n > 0) {
        r *= n;
        n--;
    }
    return r;
}
```

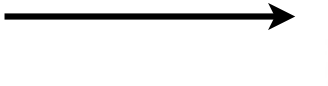
"Metal"

How does it all work????

Metal

Machine Code (bits)

```
01011111111111101110100000011111
11000111110000001101110111011111
01111101111110111101111111111111
11011111110000000000000000000000
11000000000000010000000000000000
00000000000000010000000000000000
01111100011110011001010110010000
10000000110000001111111111111111
11000000000000000010110110101010
01000000000000000000000001000001
01000000010100000000000000000000
```



"Metal"

CPU is low-level (a glorified calculator)

Assembly Code

```
fact:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -4(%rbp)
    movl    $1, -8(%rbp)
L1:
    cmpl    $0, -4(%rbp)
    jle     L2
    movl    -4(%rbp), %eax
    imull   -8(%rbp), %eax
    movl    %eax, -8(%rbp)
    mov     -4(%rbp), %eax
    addl    $-1, %eax
    movl    %eax, -4(%rbp)
    jmp     L1
L2:
    movl    -8(%rbp), %eax
    popq    %rbp
    retq
```

Machine Code

```
01011111111111101110100000111
11000111110000001101110111011111
01111101111110111101111111111111
11011111100000000000000000000000
11000000000000010000000000000000
00000000000000010000000000000000
01111100011110011001010110010000
10000000110000001111111111111111
11000000000000000001011011010100
01000000000000000000000100000000
01000000101000000000000000000000
```

"Human" readable machine code

High Level Programming

Source Code

```
int fact(int n) {
    int r = 1;
    while (n > 0) {
        r *= n;
        n--;
    }
    return r;
}
```

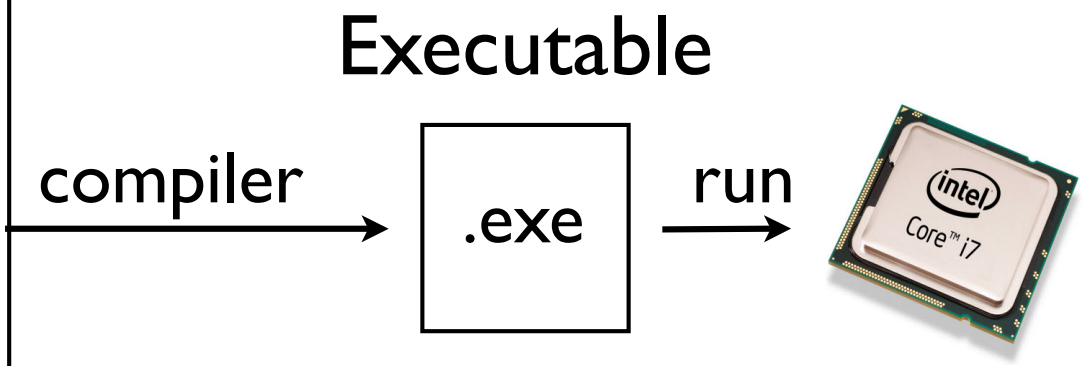
```
fact:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -4(%rbp)
    movl    $1, -8(%rbp)
L1:
    cmpl    $0, -4(%rbp)
    jle     L2
    movl    -4(%rbp), %eax
    imull   -8(%rbp), %eax
    movl    %eax, -8(%rbp)
    mov     -4(%rbp), %eax
    addl    $-1, %eax
    movl    %eax, -4(%rbp)
    jmp     L1
L2:
    movl    -8(%rbp), %eax
    popq    %rbp
    retq
```

"Human understandable" programming

Compilers

Source Code

```
int fact(int n) {
    int r = 1;
    while (n > 0) {
        r *= n;
        n--;
    }
    return r;
}
```



Compiler: A tool that translates a high-level program into bits that can actually execute.

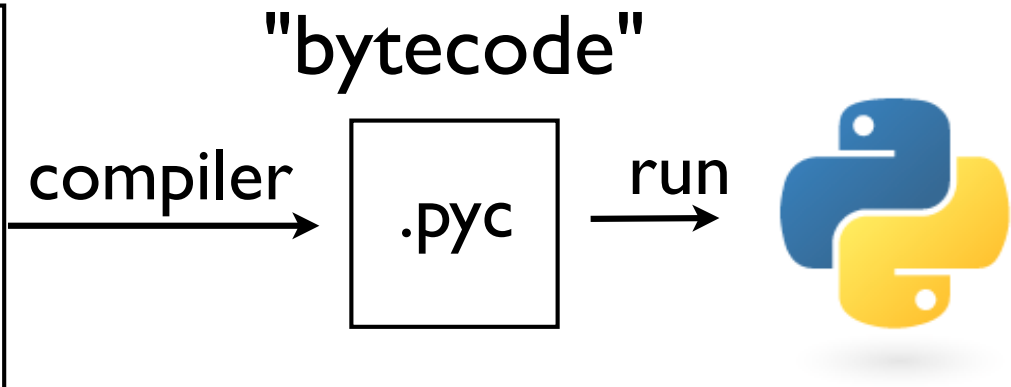
Demo: C Compiler

```
#include <stdio.h>

int fact(int n) {
    int r = 1;
    while (n > 0) {
        r *= n;
        n--;
    }
    return r;
}

int main() {
    int n;
    for (n = 0; n < 10; n++) {
        printf("%i %i\n", n, fact(n));
    }
    return 0;
}
```

Virtual Machines

Source Code

```
def fact(n):
    r = 1
    while n > 0:
        r *= n
        n -= 1
    return r;
```



Many languages run virtual machines that work like high level CPUs (Python, Java, etc.)

Demo: Python Bytecode

```
def fact(n):
    r = 1
    while n > 0:
        r *= n
        n -= 1
    return r
```

View bytecode:

```
>>> fact.__code__.co_code
b'd\x01}\x01x\x1c|\x00d\x02k\x04r |\x01|\x009\x00}\x01|
\x00d\x018\x00}\x00q\x06W\x00|\x01S\x00'
>>> import dis
>>> dis.dis(fact)
...
```

Transpilers

Source Code

```
int fact(int n) {
    int r = 1;
    while (n > 0) {
        r *= n;
        n--;
    }
    return r;
}
```

translate

Source Code

```
def fact(n):
    r = 1
    while n > 0:
        r *= n
        n -= 1
    return r
```

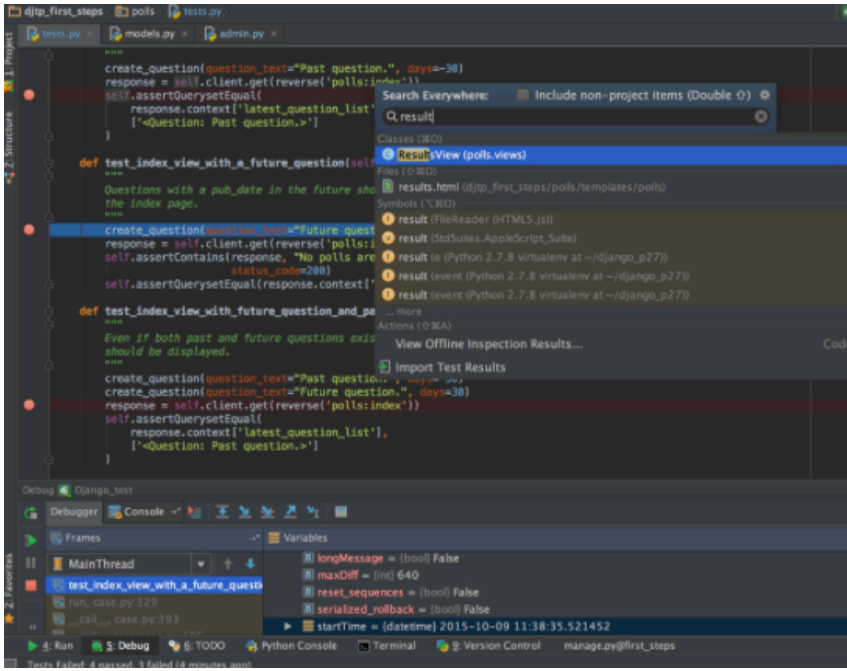
- Translation to a different language
- Example: Compilation to Javascript, C, etc.

Other Tooling

Source Code

```
int fact(int n) {
    int r = 1;
    while (n > 0) {
        r *= n;
        n--;
    }
    return r;
}
```

checking/
analysis



- Code checking (linting, formatting, etc.)
- Refactoring, IDE tool-tips, etc.

Background

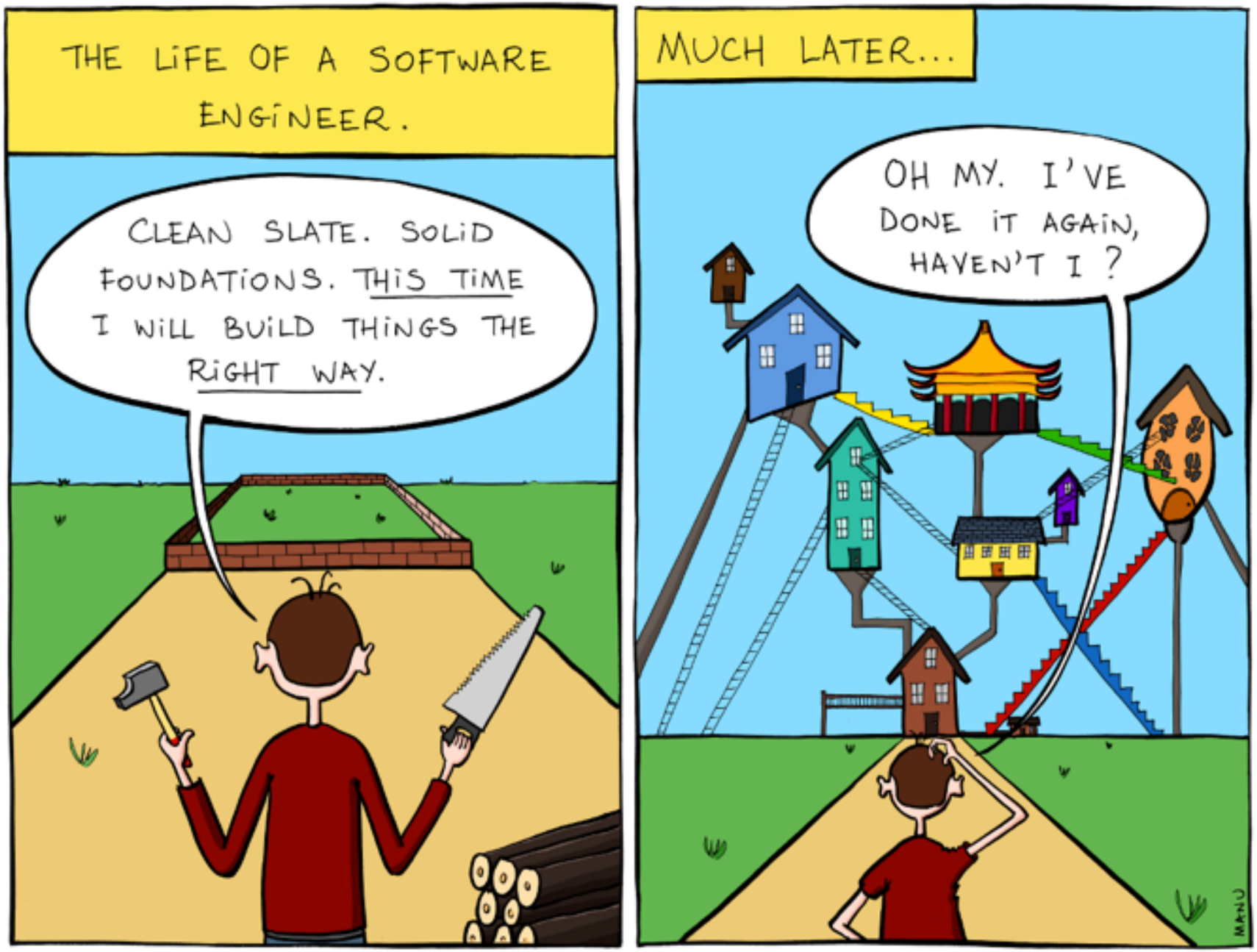
- Compilers are one of the most studied topics in computer science
- Huge amount of mathematical theory
- Interesting algorithms
- Programming language design/semantics
- The nature of computation itself

Compilers are Current!

- Flurry of new languages (Rust, Go, Julia, etc.)
- New tech (WebAssembly, The Cloud, etc.)
- Opinion: We're in a period of transition

Compiler Writing is Fun!

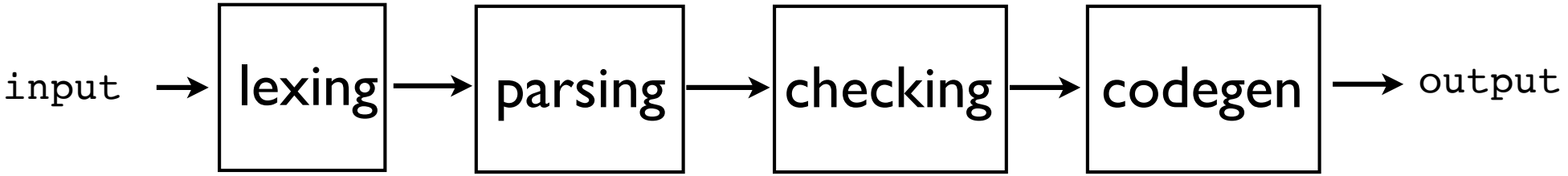
- It's one of the most complex programming projects you will likely undertake
- Many layers of abstraction (and often tooling)
- Involves just about every topic in computer science (algorithms, hardware, etc.)
- "Hey, I wrote a compiler!!!!"



http://www.bonkersworld.net

Behind the Scenes

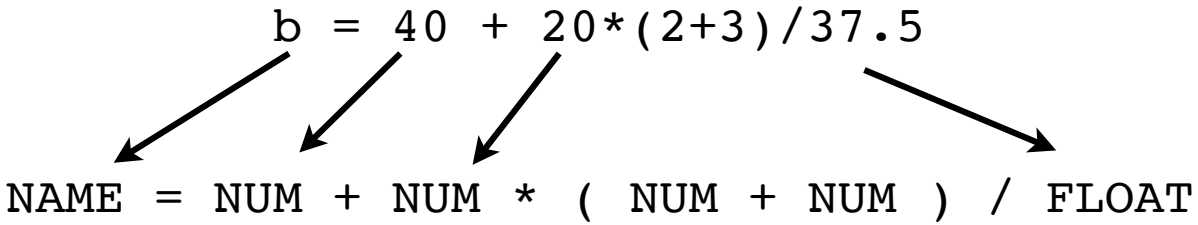
- Compilers are usually constructed as a workflow



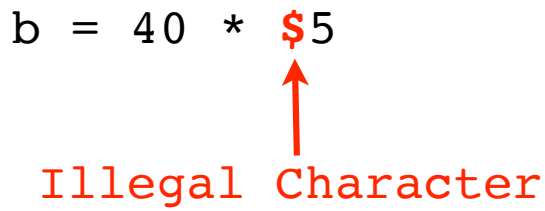
- Each responsible for a different problem.

Lexing

- Splits input text into words called tokens



- Identifies valid words, detects illegal input



- Analogy: Take text of a sentence and break it down into valid words from the dictionary

Parsing

"A ship shipping ship shipping shipping ships"

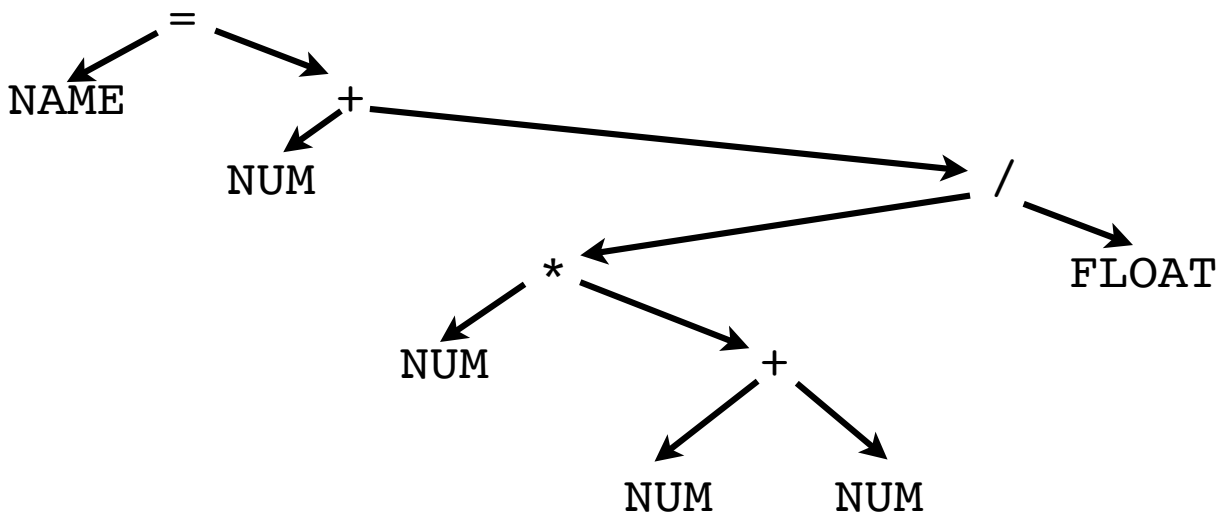
Parsing

- Verifies that input is grammatically correct

```
b = 40 + 20*(2+3)/37.5
```

- Builds a data structure representing the input

```
NAME = NUM + NUM * ( NUM + NUM ) / FLOAT
```

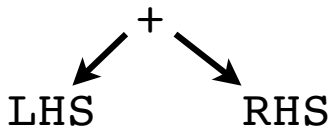


Type Checking

- Enforces rules (aka, the "legal department")

```
b = 40 + 20*(2+3)/37.5      (OK, Maybe?)
c = 3 + “hello”             (TYPE ERROR)
d[4.5] = 4                  (BAD INDEX)
```

- Example: + operator




```
1. LHS and RHS must be compatible types
2. The type must implement +
3. The result type is the same as both operands
```

Code Generation

- Generation of "output code":

```
b = 40 + 20*(2+3)/37.5
```

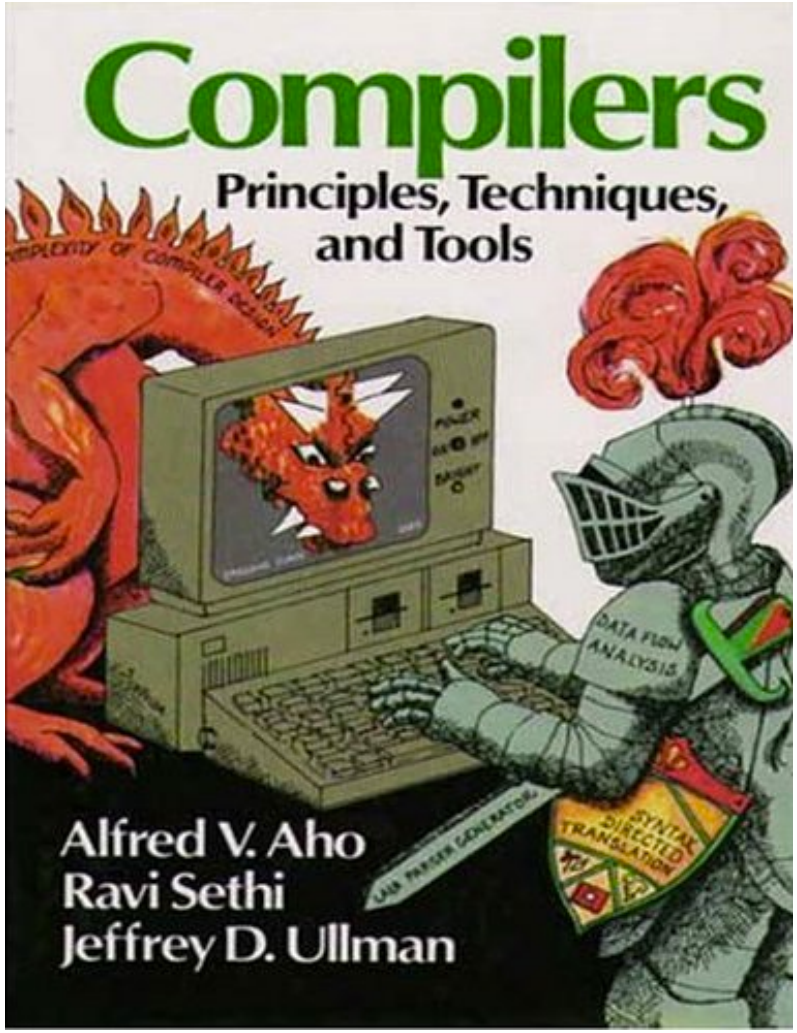
```
LOAD R1, 40
LOAD R2, 20
LOAD R3, 2
LOAD R4, 3
ADD  R3, R4, R3   ; R3 = (2+3)
MUL  R2, R3, R2   ; R2 = 20*(2+3)
LOAD R3, 37.5
DIV  R2, R3, R2   ; R2 = 20*(2+3)/37.5
ADD  R1, R2, R1   ; R1 = 40+20*(2+3)/37.5
STORE R1, “b”
```

- Wide variety of possible "outputs" here

Why Write a Compiler?

- Doing so demystifies a huge amount of detail about how computers and languages work
- It makes you a better developer
- It's a challenging software engineering project

Books



- The "Dragon Book"
- Quite challenging
- Typically taught to graduate CS students

Teaching Compilers

- Mathematical approach
    - Lots of formal proofs, algorithms, possibly some implementation in a functional language (LISP, ML, Haskell, etc.)
- Systems approach (our approach)
    - Software design, computer architecture, implementation of a compiler in C, C++, Java. More intuitional approach to theory.

Heresy!

- Many compiler courses are taught in a narrative that follows the structure of a compiler
- Lexing -> Parsing -> Checking -> CodeGen
- Each stage builds upon the previous stage
- I am NOT going to follow that path
- Instead: The "Star Wars" narrative

Now

"WHAT is happening?!?!?"

Now

understanding the problem

{

Programming

- Data Model
- Evaluation
- Semantics

Day 1

Now

Day 1

understanding the problem

Programming

- Data Model
- Evaluation
- Semantics

parsing ⟵ rest of week ⟶ code generation

Tips

- We are going to be writing a lot of code.
- I will be trying to point you in the right direction and to set the pace.
- It's a green-field project. You get little code!

Making Progress

- Parts of the project are tricky
- It's not always necessary to solve all problems at once
- You're not being graded on "coding style."
- I will push you to more forward and come back to various problems later (it's okay)

Teaching Philosophy

- I'm a proponent of "learn by doing"
- Much of what we cover in this course will be "discovered" in the process of building the project (as opposed to watching in slides)
- There is often no one "right" way to solve a problem (many possible solutions, with nuance)

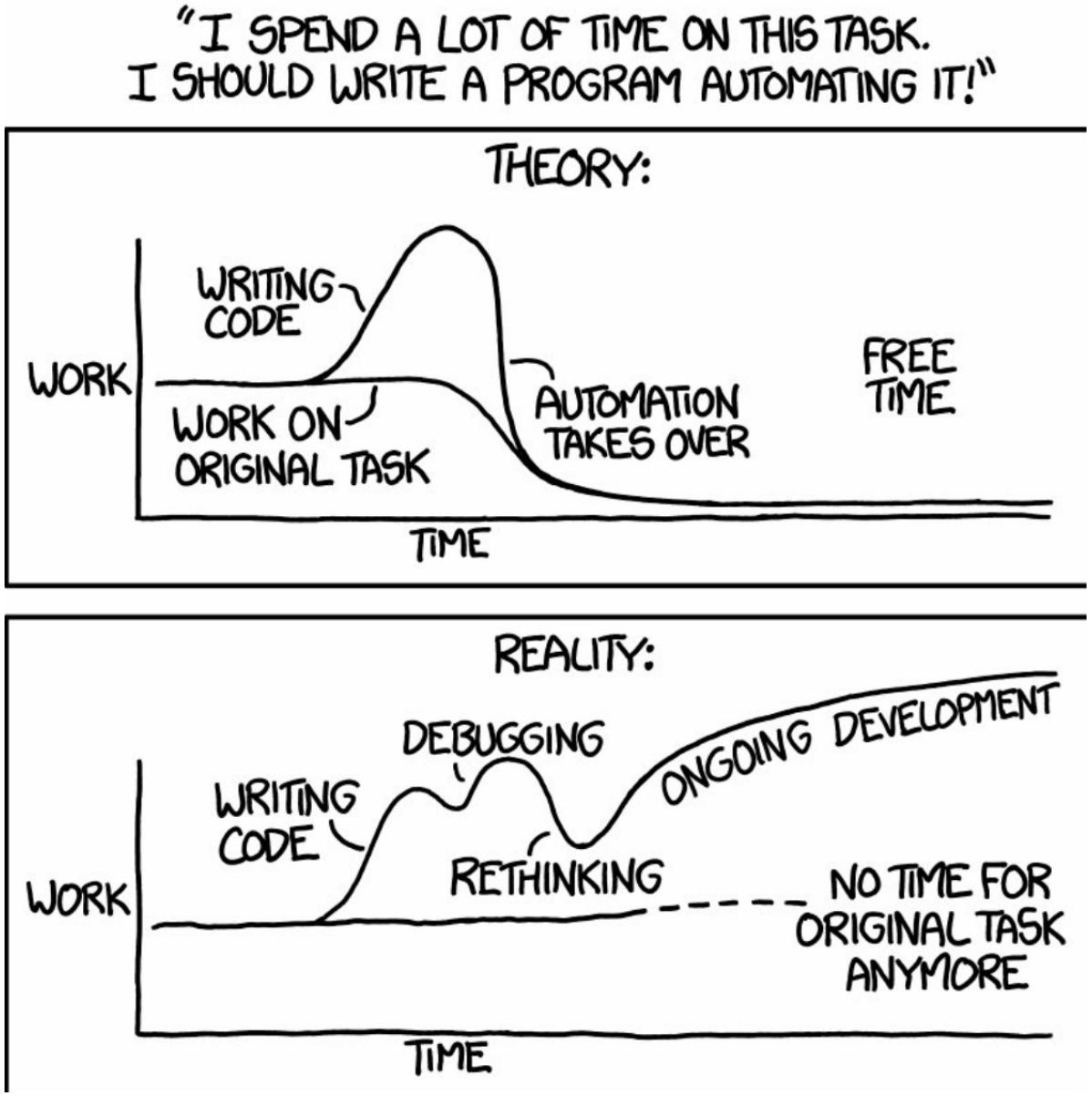
Caution

- For success, you need as few distractions as possible (work, world cup, child birth, global pandemics, etc.)

Common Project Fails

- Testing: Write tests. Think about tests. Test what you can. Do NOT write a test framework.
- DRY: There is a lot of repetition. It may be faster to just repeat code than to figure out how to not.
- Clever Code: Yes, you could use metaclasses and decorators. Or you could write a compiler.
- Overthinking: It's easy to over-architect (i.e., OO design). Keep it simple. Refactor later.

Common Project Fails

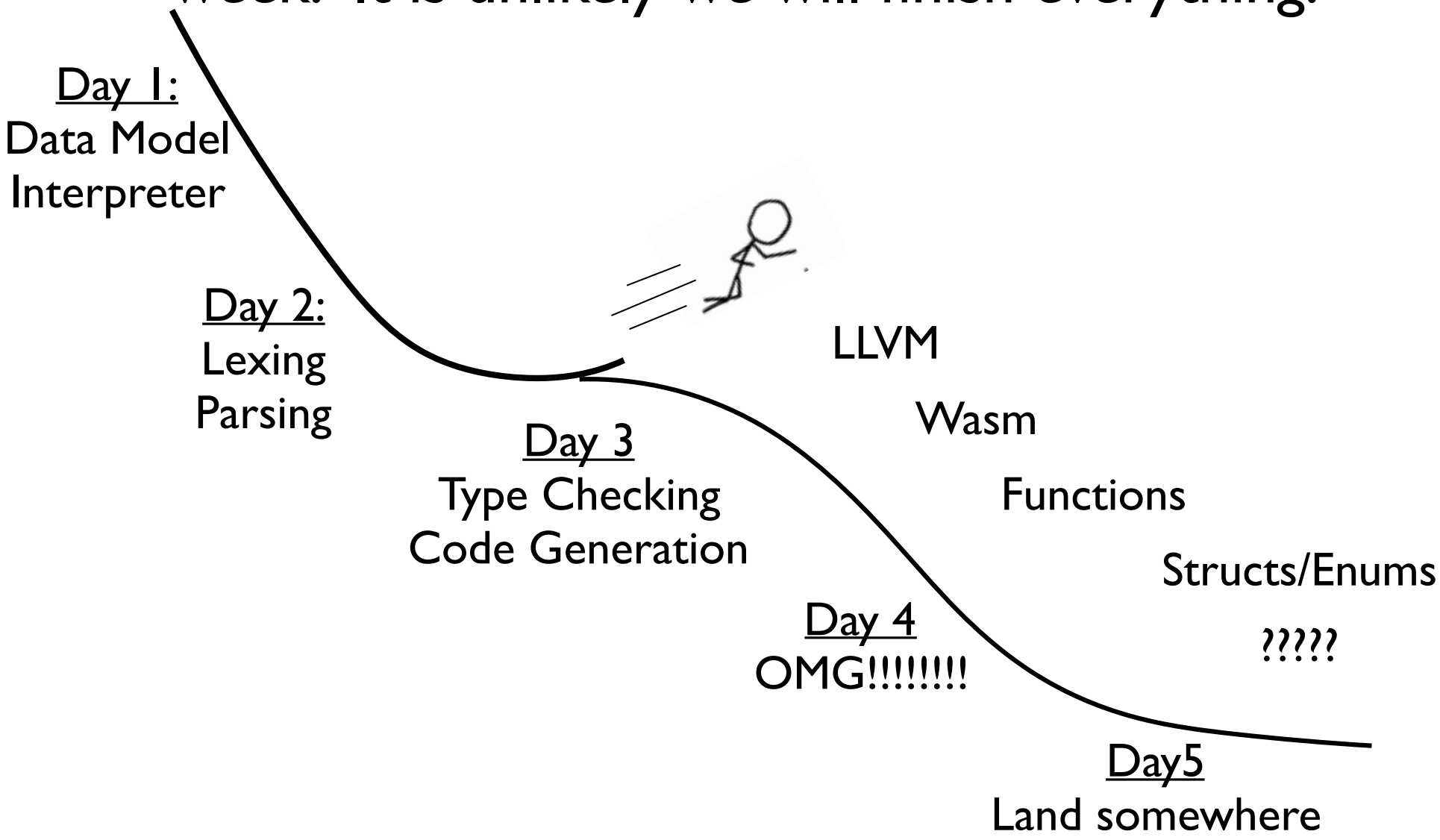


Common Project Fails

- Silent Suffering: Get my attention if something is inexplicably broken. I have written the compiler about a dozen different ways. I have already suffered through the debugging and may have a quick answer ("oh, that's likely caused by X!").
- Going too fast: It's not a race. Spending extra time to solidify your understanding usually pays off later on.

A Final Note

- The project is designed to keep you busy the entire week. It is unlikely we will finish everything.



Let's Write a Compiler ...